UNIVERSIDAD DE JAÉN

Universidad de Jaén

Práctica 1

Algoritmos de búsqueda local y metaheurísticas basadas en trayectorias

UNIVERSIDAD DE JAÉN

Metaheurísticas

GRUPO 2

Jesús Morales Villegas	---	@Jmv00037	---	77770715X
Jesús Manzano Álvarez	---	@Jma00068	---	20887601J

1. Objetivo de la práctica

El objetivo de esta práctica es estudiar el funcionamiento de Algoritmos de Técnicas de Búsqueda Local y Metaheurísticas basadas en trayectorias.

Para ello tendremos que implementar varios algoritmos como: Algoritmo de búsqueda local del mejor y Algoritmo de multiarranque con conceptos tabú.

2. Descripción del problema

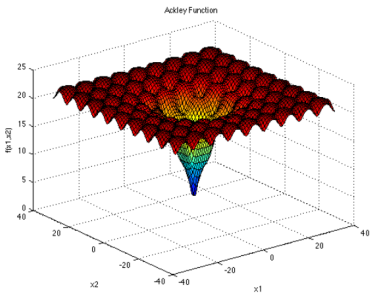
Disponemos de 10 funciones matemáticas, las cuales, son habitualmente utilizadas para la evaluación de algoritmos metaheurísticos.

Estas se caracterizan por recibir un vector, con una cierta longitud (dimensión). Este vector es construido con valores entre un rango determinado, el cual, viene definido en la descripción de dicha función. Finalmente, nos devolverá un valor dentro del dominio definido.

Todas estás funciones tienen un óptimo global y rango determinado, por tanto, vamos a exponer concretamente las principales características de cada una.

2.1 Función de Ackley

La función suele evaluarse sobre el hipercubo $x_i \in [-32.768, 32.768]$, para todo i = 1, …, d, aunque también puede estar restringida a un dominio más pequeño.



Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \ldots, 0)$$

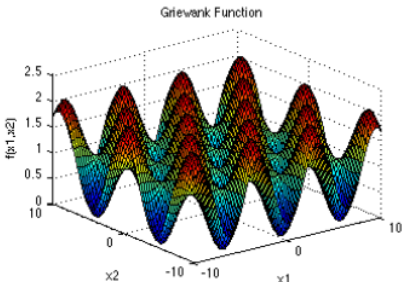
$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

2.2 Función de Griewank

La función suele evaluarse sobre el hipercubo $x_i \in [-600, 600]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \ldots, 0)$$



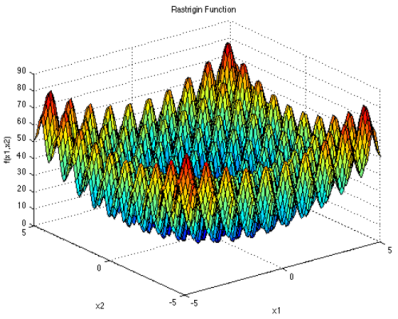
$$f(\mathbf{x}) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

2.3 Función de Rastrigin

La función suele evaluarse sobre el hipercubo $x_i \in [-5.12, 5.12]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumpe la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \ldots, 0)$$



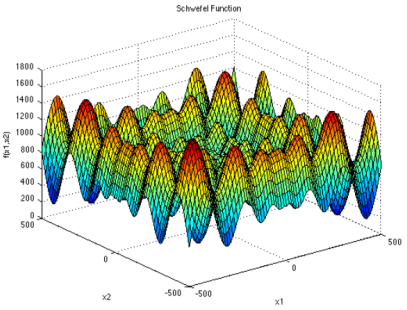
$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} [x_i^2 - 10\cos(2\pi x_i)]$$

2.4 Función de Schewefel

La función suele evaluarse sobre el hipercubo $x_i \in [-500, 500]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (420.9687, \ldots, 420.9687)$$



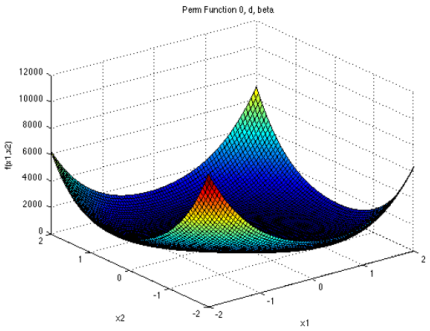
$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

2.5 Función permanente 0, D, Beta

La función suele evaluarse sobre el hipercubo $x_i \in [-d, d]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumpe la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = \left(1, \tfrac{1}{2}, \ldots, \tfrac{1}{d}\right)$$



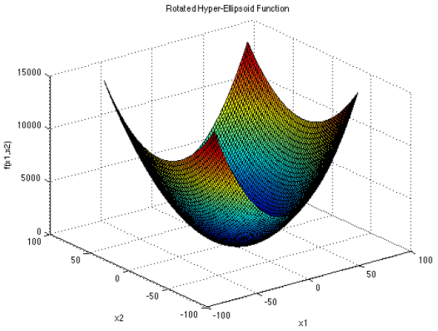
$$f(\mathbf{x}) = \sum_{i=1}^{d} \left(\sum_{j=1}^{d} (j+\beta) \left(x_j^i - \frac{1}{j^i} \right) \right)^2$$

2.6 Función Rotated Hyper-Ellipsoid

La función suele evaluarse sobre el hipercubo $x_i \in [-65.536, 65.536]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \ldots, 0)$$



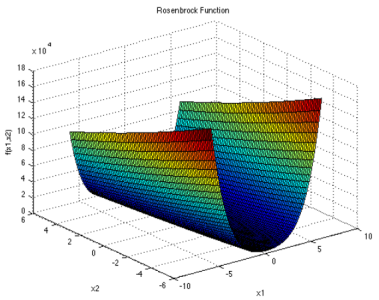
$$f(\mathbf{x}) = \sum_{i=1}^{d} \sum_{j=1}^{i} x_j^2$$

2.7 Función de Rosenbrock

La función suele evaluarse sobre el hipercubo $x_i \in [-5, 10]$, para todo i = 1, …, d, aunque puede restringirse al hipercubo $x_i \in [-2.048, 2.048]$, para todo i = 1, …, D.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (1, \ldots, 1)$$



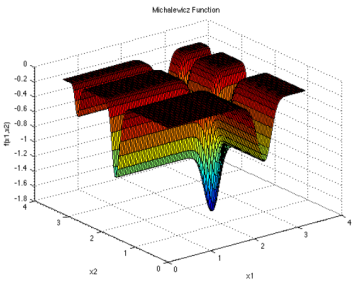
$$f(\mathbf{x}) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

2.8 Función de Michalewicz

La función suele evaluarse sobre el hipercubo $x_i \in [0, \pi]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumplen las siguientes condiciones, dependiendo del valor de la dimensión:

$$\text{at } d = 2\text{: } f(\mathbf{x}^*) = -1.8013, \text{ at } \mathbf{x}^* = (2.20, 1.57)$$
$$\text{at } d = 5\text{: } f(\mathbf{x}^*) = -4.687658$$
$$\text{at } d = 10\text{: } f(\mathbf{x}^*) = -9.66015$$



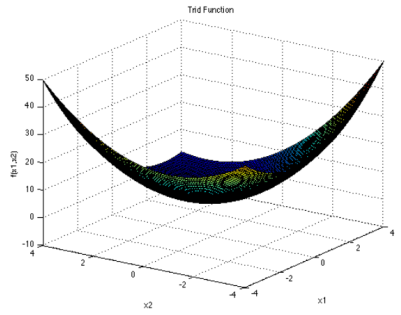
$$f(\mathbf{x}) = -\sum_{i=1}^{d} \sin(x_i) \sin^{2m}\left(\frac{ix_i^2}{\pi}\right)$$

2.9 Función de Trid

La función suele evaluarse sobre el hipercubo $x_i \in [-d^2, d^2]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = -d(d+4)(d-1)/6, \text{ at } x_i = i(d+1-i), \text{ for all } i = 1, 2, \ldots, d$$



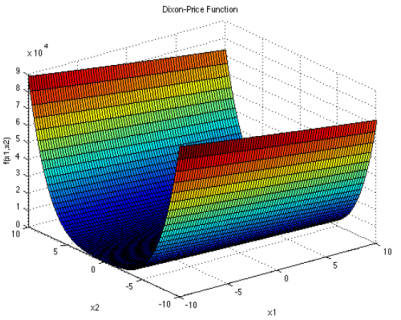
$$f(\mathbf{x}) = \sum_{i=1}^{d} (x_i - 1)^2 - \sum_{i=2}^{d} x_i x_{i-1}$$

2.10 Función de Dixon-price

La función suele evaluarse sobre el hipercubo $x_i \in [-10, 10]$, para todo i = 1, …, d.

Su mínimo global se da cuando se cumple la siguiente condición:

$$f(\mathbf{x}^*) = 0, \text{ at } x_i = 2^{-\frac{2^i - 2}{2^i}}, \text{ for } i = 1, \ldots, d$$

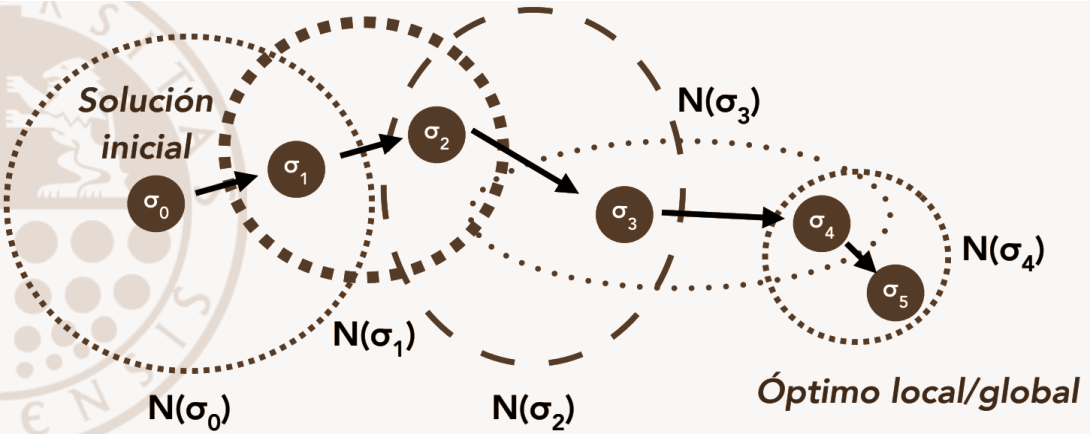


$$f(\mathbf{x}) = (x_1 - 1)^2 + \sum_{i=2}^{d} i \left(2x_i^2 - x_{i-1}\right)^2$$

3. Algoritmos basados en trayectorias

Los algoritmos basados en trayectorias son la base de muchos de los métodos utilizados actualmente en problemas de optimización.

Estos algoritmos se pueden ver como un proceso iterativo que parte de una solución inicial (aleatoria o en base a un criterio) y la mejora con el tiempo, realizando modificaciones locales.



Básicamente, empieza con una solución inicial y busca en su vecindad por una mejor solución. Si la encuentra, reemplaza su solución actual por la nueva y continua con el proceso, hasta que el algoritmo finaliza (la condición de parada depende del algoritmo en específico que utilicemos).

Además, en nuestro caso hemos añadido algunas modificaciones a las características principales de una búsqueda local del mejor (Apartado 3.1).

```
INICIO
    GENERA(Solución Inicial)
    Solución Actual <- Solución Inicial;
    Mejor Solución <- Solución Actual;
    REPETIR
        Solución Vecina <- GENERA_VECINO(Solución Actual);
        SI Acepta(Solución Vecina)
            ENTONCES Solución Actual <- Solución Vecina;
        SI Objetivo(Solución Actual) es mejor que Objetivo(Mejor Solución)
            ENTONCES Mejor Solución <- Solución Actual;
    HASTA(Criterio de parada);
    DEVOLVER(Solución Actual);
FIN
```

3.1 Esquema general de la búsqueda local del mejor

Hasta el momento hemos visto, la estructura o esquema general de las búsquedas por entornos. Para el caso de la búsqueda local del mejor, vamos a aplicar ciertas restricciones o condiciones que el enunciado del problema nos indicaba.

En primer lugar, vamos a exponer como es el pseudocódigo de este algoritmo, y definiremos ciertas variables que aparecen para que todo quede más claro.

```
INICIO
    GENERA(Solución Actual)
    Mejor Vecino <- Solución Actual;
    REPETIR i=1 hasta n_iteraciones && mejore
        REPETIR j=1 hasta k vecinos
            REPETIR k=1 hasta d
                aleatorio <- GENERAR_ALEATORIO[0, 1]
                SI aleatorio < PROB_ESCOGERLO
                    ENTONCES vecino[k] = aleatorio[Solución Actual[k] -MARGEN, MARGEN];
                ELSE
                    ENTONCES vecino[k] = Solución Actual[k];
            FIN REPETIR
            SI vecino es mejor que Mejor Vecino
                ENTONCES Mejor Vecino <- vecino;
        FIN REPETIR
        SI Mejor Vecino es mejor que Solución Actual
            ENTONCES Solución Actual <- Mejor Vecino;
                     Actualizamos las variables mejore y n_iteraciones;
    FIN REPETIR
    DEVOLVER Solución Actual
FIN
```

- **Solución Actual:** Es un vector de tamaño d, donde se guarda la solución inicial y a partir de ese momento la última solución a la que nos hemos movido.

- **Vecino:** Es un vector de tamaño d, el cual, se encarga de almacenar todos los valores que vamos generando.

- **Mejor Vecino:** Es un vector de tamaño d, en el cual, se almacena siempre el mejor vecino de todo el entorno generado.

- **Prob_Escogerlo:** Es un valor el cual está comprendido entre 0 y 1, indicando la probabilidad necesaria para que el elemento que ha sido generado aleatoriamente sea seleccionado.

- **Margen:** Es un valor comprendido entre 0 y 1, indicando el porcentaje que hay de aumento en el rango con respecto al valor almacenado en el vector solución actual.

- **N_Iteraciones:** Es un valor que indica el número de movimientos que se han producido durante la ejecución del algoritmo.

En nuestro caso con respecto a este tipo de algoritmo vamos a implementar dos modificaciones muy parecidas.

Todo va a depender del número de vecinos que se generan, en el primer caso (BL3) siempre se generarán 3 vecinos en cada entorno y en el segundo caso (BLK) se genera de forma aleatoria entre 4 y 10 cada vez que se crea un nuevo entorno de vecinos (esto se puede cambiar en el fichero config.txt en las variables NumVecinosMin y NumVecinosMax).

3.2 Esquema general del algoritmo multiarranque con conceptos tabú

Estos algoritmos también son unas búsqueda la cuales se basan en los algoritmos basados en trayectorias. Por tanto, la idea es muy similar al anterior, aunque se le añade la posibilidad de hacer movimientos de empeoramiento (siempre se coge la mejor solución de las generadas aunque sea peor que el anterior movimiento) y además conceptos como la memoria a corto y largo plazo.

La memoria a corto plazo almacena todas los últimos movimientos, lo cual nos ayuda a no movernos a soluciones recientemente exploradas.

En el caso de la memoria a largo plazo almacena todos los movimientos realizados durante la ejecución, por tanto, nos ayuda a analizar cuales son las zonas del espacio más y menos visitadas, para poder hacer intensificación o diversificación.

Ahora vamos a ver el pseudocódigo de este tipo de algoritmos y explicar cuales son las variables que tiene más relevancia (algunas ya están explicadas anteriormente).

- **Mejor Solución Global:** Almacena la mejor solución encontrada a lo largo de toda la ejecución. Esta solución nunca puede ir a peor.
- **Lista Tabú:** Es una lista que almacena todas las últimas t soluciones.
- **Movimientos Tabú:** Es una lista que almacena todos los movimientos que se han producido por cada uno de los cambios de solución.
- **Oscilación Estratégica:** Indica el valor para deducir entre hacer el proceso de intensificación o el proceso de diversificación. Este valor está comprendido entre [0, 1].

```
INICIO
    GENERA(Solución Actual)
    Mejor Vecino <- Solución Actual;
    REPETIR i=1 hasta n_iteraciones
        REPETIR j=1 hasta k vecinos
            REPETIR k=1 hasta d
                aleatorio <- GENERAR_ALEATORIO[0, 1]
                SI aleatorio < PROB_ESCOGERLO
                    ENTONCES vecino[k] = aleatorio[Solución Actual[k] -MARGEN, MARGEN];
                ELSE
                    ENTONCES vecino[k] = Solución Actual[k];
            FIN REPETIR
            SI vecino es mejor que Mejor Vecino y no es tabú
                ENTONCES Mejor Vecino <- vecino;
        FIN REPETIR
        Movimientos Tabú <- obtener diferencia movimientos entre Mejor Vecino y Solución Actual
        SI Lista Tabú es de tamaño t
            ENTONCES eliminamos el primer elemento
        Lista Tabú <- Mejor Vecino;
        Solución Actual <- Mejor Vecino;
        AGREGAR_SOLUCION_LARGO_PLAZO(Solución Actual);
        SI Solución Actual es mejor que la Mejor Solución Global
            ENTONCES Mejor Solución Global <- Solución Actual;
        Actualizamos las variables contReinicio y n_iteraciones;
        SI contReinicio pasa el intervalo de reinicialización
            ENTONCES aleatorio <- GENERAR_ALEATORIO[0, 1]
                SI aleatorio < Oscilación Estratégica
                    ENTONCES diversificamos;
                ELSE intensificamos;
    FIN REPETIR
    DEVOLVER Mejor Solución Global;
FIN
```

4. Operadores utilizados

Vamos a describir, los operadores que tienen cierta relevancia a la hora de ejecutar nuestros algoritmos. Para ello vamos a dividirlos en dos apartados.

4.2 Parámetros de configuración

En el archivo de configuración encontramos diferentes parámetros que usaremos en los diferentes algoritmos. Estos se extraerán del archivo “config.txt”, en el que añadiremos las diferentes constantes a usar en las diferentes ejecuciones.

- **ArrayList <String> archivos:** En el almacenamos los nombres de los ficheros donde se guardan los datos a utilizar en el problema.

- **ArrayList <String> algoritmos:** Almacena el nombre de todos los algoritmos que queremos ejecutar.

- **ArrayList <Long> semillas:** Almacena todas las semillas que vamos utilizar en las distintas ejecuciones. En nuestro caso, vamos a generar 5 semillas partiendo de una original 20887601(las siguientes cuatro se crean haciendo permutaciones de la original), con lo cúal los resultados con estas semillas siempre deberían de ser los mismos.

- **Dimension:** Almacena el valor del tamaño del vector (d, que por defecto es 10) que reciben nuestras funciones de evaluación.

- **Iteraciones:** Almacena el valor de N_Iteraciones, explicado en el esquema de búsqueda general. Por defecto, este valor está establecido a 1000.

- **ProbCambio:** Almacena el valor de Prob_Escogerlo, explicado en el esquema de la búsqueda general. Por defecto, este valor está establecido al 30 % => 0.3

- **DiffRango:** Almacena el valor de Margen, explicado en el esquema de la búsqueda general. Por defecto, este valor está establecido al 10 % => 0.1

- **NumDecimales:** Almacena el valor del número de decimales que como máximo van a tener todos los valores obtenidos (Podemos ser más exactos o menos). Tal y como lo hemos implementado nosotros, siempre redondeamos, es decir, no truncamos los valores.

- **NumVecinos:** Almacena el número de vecinos por defecto, aunque este se puede modificar en tiempo de ejecución, dependiendo del algoritmo que se ejecute. Por defecto, el valor es 3.

- **NumVecinosMin:** Dentro del rango que puede variar el número de vecinos, NumVecinosMin, indica el valor (incluido dicho valor) mínimo que puede generar. Este valor solo se tiene en cuenta para los algoritmos que lo necesiten.

- **NumVecinosMax:** Dentro del rango que puede variar el número de vecinos, NumVecinosMax, indica el valor (incluido dicho valor) máximo que puede generar. Este valor solo se tiene en cuenta para los algoritmos que lo necesiten.

- **RangoTabu:** Indica el valor del porcentaje que como máximo puede diferir un valor para que se considere dentro del rango. Por defecto, tiene un valor del 1%.

- **TendenciaTabu:** Indica el tamaño máximo (t) que puede tener la lista tabú (lista que almacena las últimas t soluciones). Por defecto, tiene un valor de 5.

- **IntervaloReinicializacion:** Indica el porcentaje de movimientos no mejorados necesarios para que se produzca una reinicialización. Por defecto, tiene un valor del 5%.

- **OscilacionEstrategica:** Indica el porcentaje de decisión entre diversificar e intensificar, cuando se produce una reinicialización. Por defecto, tiene un valor del 50%.

4.3 Métodos utilizados

Vamos a exponer algunos de los métodos más utilizados para la ejecución de nuestro programa.

- **double evaluar (double[] vector):** Este método es el principal de todas las funciones matemáticas implementadas. Aquí es donde se calcula un valor dado un vector y posteriormente se devuelve dicho valor.

- **double getRango_min ():** Este método devuelve el valor del rango mínimo en el que se evalúa la función, este dependerá de cada función como hemos visto es la explicación de cada una de ellas.

- **double getRango_max ():** Este método devuelve el valor del rango máximo en el que se evalúa la función, este dependerá de cada función como hemos visto es la explicación de cada una de ellas.

- **double [] busqueda (Evaluador exe, long semilla, TablaDatos datos):** Este método es principal de cada una de las búsquedas que se han implementado, ya que contiene el código propiamente dicho. Para ello recibe la función con la que queremos evaluar (exe), la semilla con la que se va a realizar toda la ejecución y finalmente la tabla de datos, donde se van a ir almacenando los datos para luego generar el fichero de salida.

- **ArrayList<double []> getSoluciones():** Devuelve una lista con todas las soluciones por las que se han ido pasando a lo largo de la búsqueda.

- **limpiaBusqueda():** Limpia el arrayList de vectores que tiene la búsqueda como atributo, esto se hace por si se vuelve a llamar a búsqueda que esté vacío.

5. Análisis de los algoritmos

Una vez implementados los distintos algoritmos, vamos a pasar a la fase de análisis. Para ello, vamos a generar una tabla de datos donde para un algoritmo, semilla y función de evaluación concreta podamos ver cúal ha sido su desviación del error y el tiempo que ha tardado en ejecutar la búsqueda.

En primer lugar, por cada algoritmo que ejecutamos con una de las semillas y evaluamos con cada una de las funciones de evaluación (las 10 explicadas anteriormente), esto nos generará un conjunto de soluciones, desde la inicial hasta la final (aunque solo nos quedamos con la final/mejor). Todas estas soluciones se irán escribiendo en unos ficheros llamados **algoritmo_semilla_nombreFuncion.txt** (en la carpeta logs), los cuales nos permitirán revisar de forma más exhaustiva cuántas y cuáles son las soluciones que han ido generando cada una de las evaluaciones, dado un algoritmo y una semilla concreta.

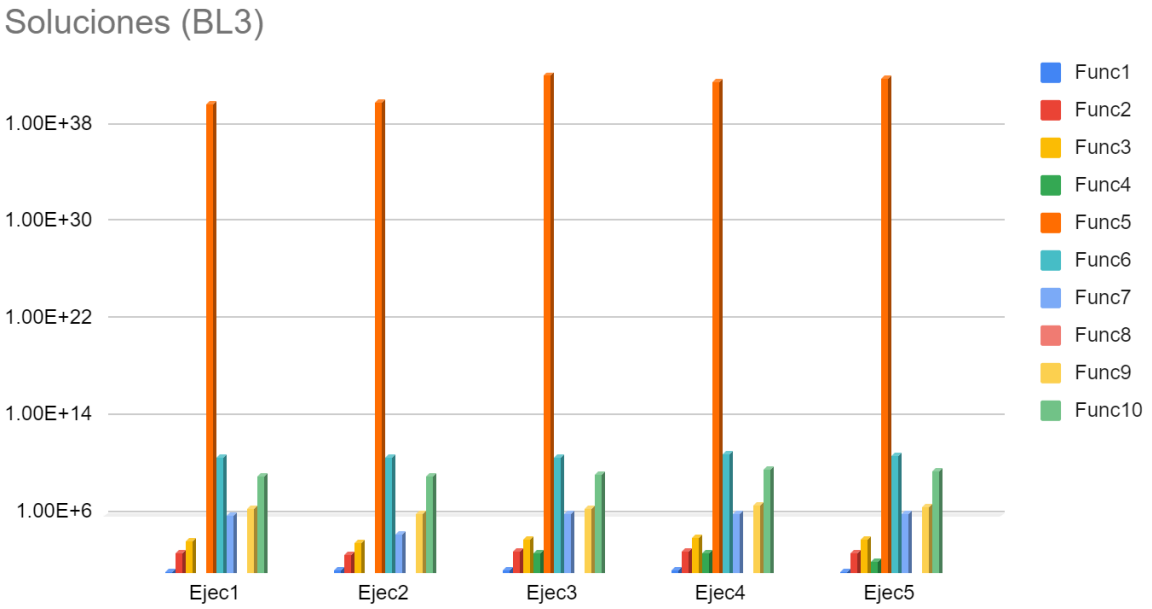
En segundo lugar, a diferencia del anterior, en este caso, vamos a generar un fichero **salida.txt**, el cual se encargue de almacenar los valores finales que irán en la tabla. Para ello, almacenamos todos estos valores en formato CSV (Comma Separated Values), que nos permitirá importar los datos en una hoja de cálculo de una forma mucho más sencilla.

Para que queden más claros los datos, primero vamos a mostrar los resultados de las distintas ejecuciones (una por cada semilla) con respecto a las **soluciones** y posteriormente con respecto al **tiempo de ejecución** (medido en microsegundo **µs**).

5.1 Búsqueda local del mejor (bl3)

Este algoritmo se caracteriza por generar siempre 3 vecinos por cada entorno que construye. Estos son los datos obtenidos:

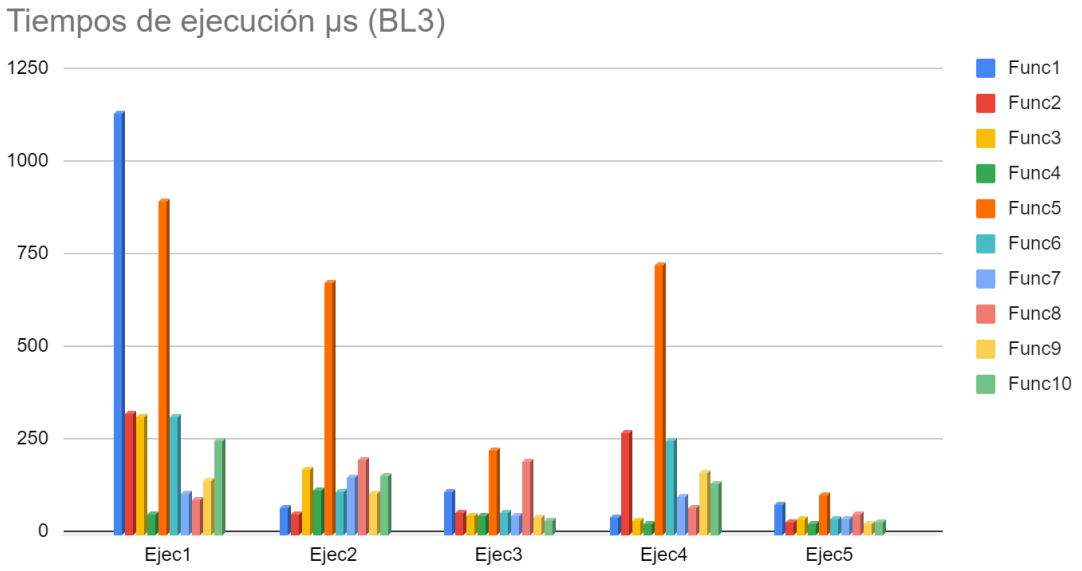
BL3	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
Ejec1	20.6582	650.0117	6865.8713	-4363.91	7.64598E+39	5.32E+10	9.11E+05	-0.1069	2.68E+06	1.48E+09
Ejec2	21.3927	530.6014	4961.4417	-3436.8754	8.48383E+39	4.89E+10	2.62E+04	-1.1774	1.28E+06	1.39E+09
Ejec3	21.1733	833.9992	8804.1511	571.0665	1.56121E+42	5.41E+10	1.06E+06	-0.0489	2.70E+06	2.24E+09
Ejec4	21.4407	799.1475	10563.3669	579.6594	4.17377E+41	8.97E+10	1.27E+06	-0.6313	5.16E+06	4.45E+09
Ejec5	20.9943	710.6256	7517.6438	144.638	1.07449E+42	6.43E+10	1.08E+06	-0.3211	3.60E+06	3.40E+09
Media	21.1318	704.8771	7742.4950	-1301.0843	6.14E+41	6.20E+10	8.70E+05	9.2030	3.08E+06	2.59E+09
Desv.	0.3195	121.4235	2099.8641	2401.8099	6.86E+41	1.65E+10	4.89E+05	0.4629	1.43E+06	1.32E+09



Hay que añadir que esta tabla está en formato logarítmico, lo cuál muestra mejor los datos más pequeños (los datos más grandes se salen de la gráfica hacia arriba).

Ahora vamos a ver todos los tiempos de ejecución medidos en microsegundos, para posteriormente hacer un estudio de cuál ha sido la mejor ejecución.

BL3	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
Ejec1	1135.1	326.9	316.8	54.6	897.6	317.2	108.3	94.3	144.8	251.2
Ejec2	72.8	55.8	176	120.5	679.6	116.4	152.6	200.8	110.2	157.4
Ejec3	114.7	58.8	49.2	51.6	227.5	58.7	50.2	198.4	47.4	36.4
Ejec4	43.7	272	34.8	27.7	726	252.3	103.3	70.1	164.5	137.1
Ejec5	78.3	33.2	41.4	29.6	107	40	39.3	52.2	27.2	33.7
Media	288.9200	149.3400	123.6400	56.8000	527.5400	156.9200	90.7400	123.1600	98.8200	123.1600
Desv.	473.7021	138.7523	122.7290	37.6703	341.4421	122.2153	46.3204	71.3663	59.8560	91.2296



Como el tiempo de ejecución son unos pocos microsegundos, podemos decir que el tiempo de ejecución es despreciable.

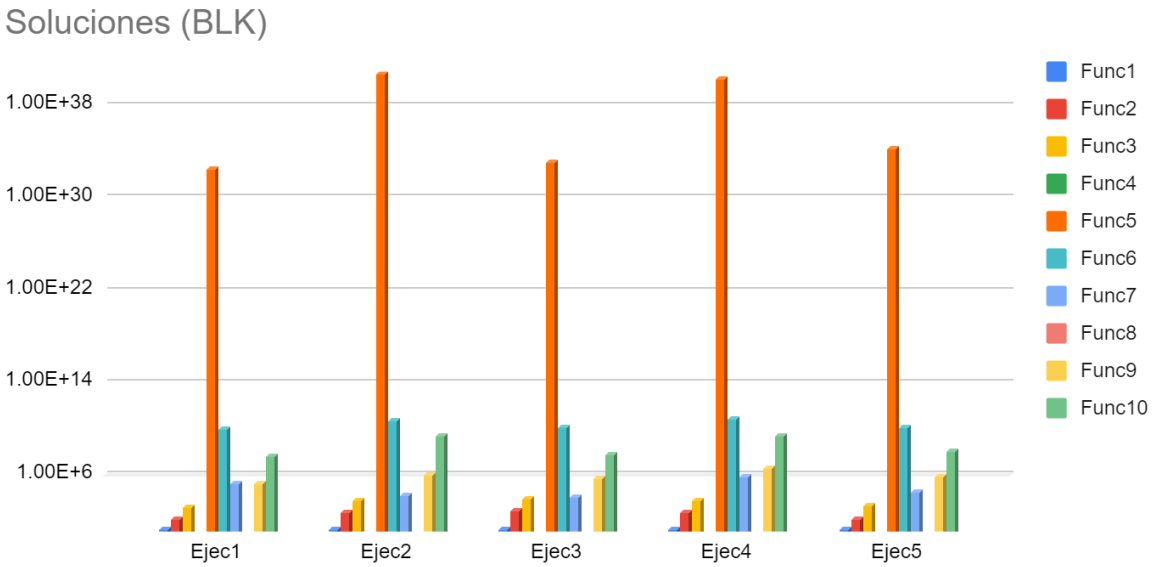
Por tanto, aunque la mejor solución con respecto al tiempo ha sido la quinta, se puede afirmar que la mejor es aquella que se acerque más a los óptimos globales, la cual ha sido la segunda ejecución, con la semilla: 8876012

5.2 Búsqueda local del mejor (blk)

Este algoritmo aunque es muy parecido al anterior, la principal diferencia es que en este caso cuando se genera un nuevo entorno no siempre se construirán 3 vecinos, es decir, se generarán un número aleatorio dentro del rango establecido por NumVecinosMin y NumVecinosMax (Por defecto 4 - 10).

BLK	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
Ejec1	20.6287	130.9096	1827.0995	-16708.1801	3.44E+32	1.04E+10	1.75E+05	-0.8727	1.69E+05	4.74E+07
Ejec2	21.284	532.0387	5612.441	-11317.5544	4.70E+40	5.01E+10	1.87E+04	-1.5181	1.06E+06	2.06E+09
Ejec3	20.9176	714.7494	7573.1653	-15192.6634	1.04E+33	1.10E+10	1.25E+04	-0.5821	4.90E+05	5.01E+07
Ejec4	20.2301	629.172	6647.5839	-4002.7486	2.01E+40	7.78E+10	7.64E+05	-1.4806	4.02E+06	2.44E+09
Ejec5	20.5029	167.9917	1948.0013	-11288.4285	1.62E+34	1.44E+10	3.10E+04	-1.0061	7.35E+05	1.03E+08

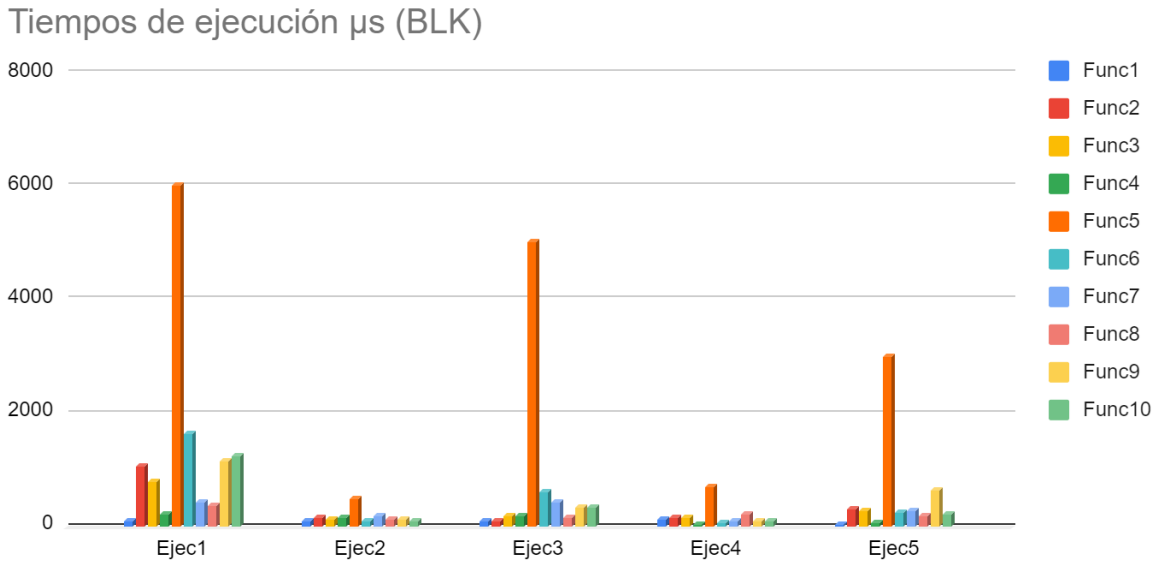
Media	20.7127	434.9723	4721.6582	-11701.9150	1.34E+40	3.27E+10	2.00E+05	8.5682	1.29E+06	9.42E+08
Desv.	0.0890	26.2210	85.4905	3832.3431	1.12E+34	2.89E+09	1.02E+05	0.4025	1.56E+06	3.91E+07



Hay que añadir que esta tabla está en formato logarítmico, lo cuál muestra mejor los datos más pequeños (los datos más grandes se salen de la gráfica hacia arriba).

Ahora vamos a ver todos los tiempos de ejecución medidos en microsegundos, para posteriormente hacer un estudio de cuál ha sido la mejor ejecución.

BLK	**Func1**	**Func2**	**Func3**	**Func4**	**Func5**	**Func6**	**Func7**	**Func8**	**Func9**	**Func10**
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
Ejec1	99.6	1069.9	778.2	217.7	6040.1	1626.7	424.2	377.2	1165.4	1237.9
Ejec2	79	140.1	119.6	158.8	485.8	84.4	182.2	131.9	116.3	84.9
Ejec3	89.4	88.4	177.8	173	5044	610.3	410.7	151.4	326.8	334.5
Ejec4	116.1	140.2	144.2	21.3	694.9	48	95.4	209.7	91.7	87.2
Ejec5	46.9	310.7	269.1	68.2	3011.6	232.9	283.2	178	643	216.8
Media	86.2000	349.8600	297.7800	127.8000	3055.2800	520.4600	279.1400	209.6400	468.6400	392.2600
Desv.	37.2645	536.8355	359.9881	105.7125	2141.4729	985.5654	99.7021	140.8557	369.3926	722.0267



En este caso el tiempo de ejecución también es despreciable, por tanto, podemos considerar que la mejor ejecución ha sido la primera, la cual utiliza la semilla 20887601.

Si comparamos los resultados con el BL3 vemos que los tiempos han aumentado, ya que este algoritmo tiene que hacer más cálculos al generar más vecinos (más lento). Aunque el tiempo sea mayor que el anterior, como estamos hablando de unos cientos de microsegundos, es algo insignificante para el tiempo total de cálculo.

Por el contrario, vemos que sí obtiene soluciones más cercanas al óptimo global en un tiempo razonable, aunque en el caso de la función 4 (Schewefel) empeora, lo cual es posible, ya que al generar un número aleatorio de vecinos, puede darse el caso que estos sean peores que los generados en el BL3, por tanto, podemos afirmar que el algoritmo BLK es mejor que el anterior.

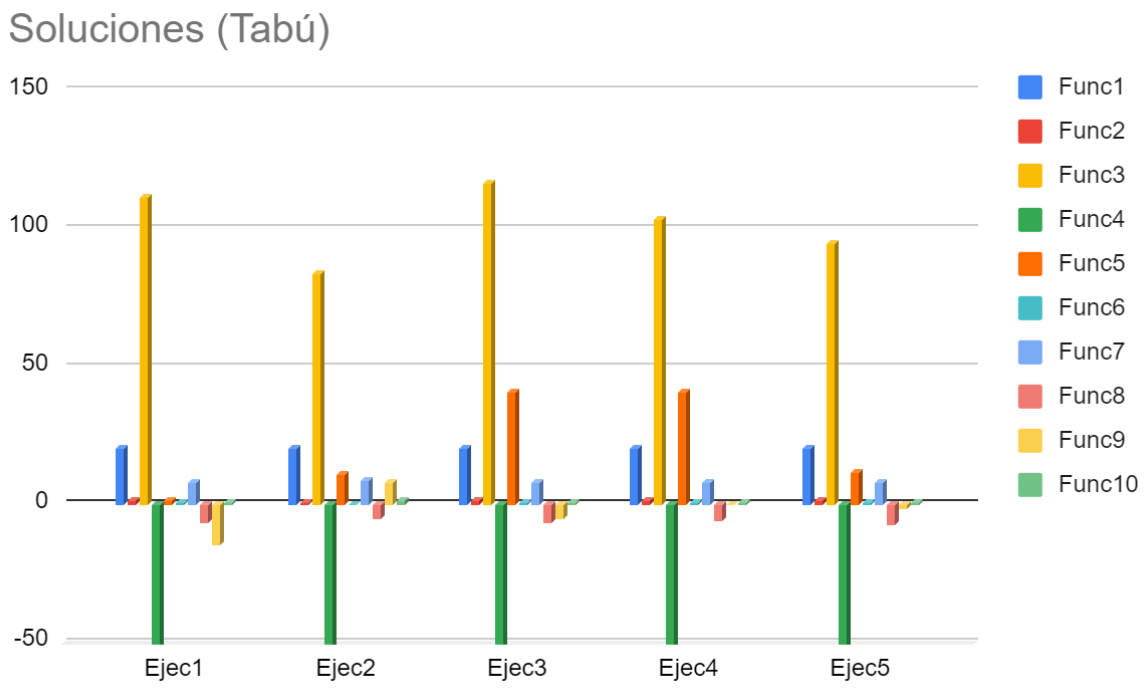
5.3 Búsqueda tabú

Este algoritmo se basa en el mejor algoritmo de los dos anteriores (BLK). Además, añade una memoria a corto plazo y largo plazo y permite movimientos de empeoramiento.

La memoria a corto plazo almacena todas los últimos movimientos con tendencia tabú de valor t. Esto ayuda a no movernos a soluciones recientemente exploradas.

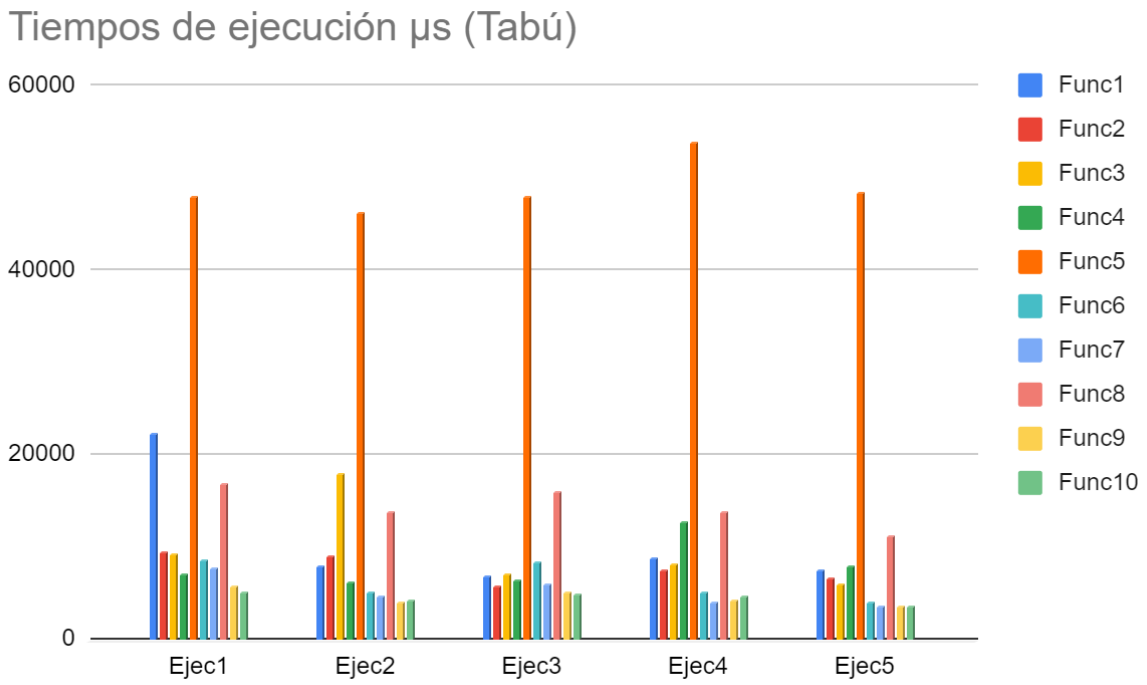
La memoria a largo plazo almacena todos los movimientos realizados durante la ejecución, por tanto, nos ayuda a analizar cuales son las zonas del espacio más y menos visitadas, para poder hacer intensificación o diversificación.

Tabú	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
Ejec1	20.3037	1.0927	111.382	-22902.003	1.3598	0.3164	8.0511	-6.4337	-14.8499	0.6667
Ejec2	20.1863	0.7365	83.4528	-14310.5189	11.1308	0.138	8.9502	-5.5781	8.0188	1
Ejec3	20.1161	1.2413	116.1528	-21568.8265	40.9934	0.9353	8.0618	-6.5795	-4.904	0.6667
Ejec4	20.2002	1.0918	103.3379	-21166.1277	40.8668	0.3656	8.0509	-5.7146	0.0537	0.6667
Ejec5	20.2164	1.2997	94.4698	-20256.9169	11.2691	0.2672	8.0511	-7.4612	-1.9427	0.6668
Media	20.20454	1.0924	101.75906	-20040.8786	21.12398	0.4045	8.23302	3.30673	207.27518	0.73338
Desv.	0.06173	0.14637	11.95873	1870.35832	7.00693	0.03479	0.00000	0.75729	8.29945	0.00007



Ahora vamos a ver todos los tiempos de ejecución medidos en microsegundos, para posteriormente hacer un estudio de cuál ha sido la mejor ejecución.

Tabú	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
Ejec1	22211.4	9556	9218.3	7016.3	47919.4	8576.4	7629.5	16790.7	5703.7	5071.7
Ejec2	7849.5	9114.8	17856.6	6297.8	46351	5121.5	4612	13743	3928.6	4313.7
Ejec3	6787.2	5694.7	6983.8	6313	47901.8	8287.8	6020.4	15955.4	5111.9	4856.1
Ejec4	8716.5	7496.1	8208.1	12783.7	53866.9	5046.2	4032.1	13818.9	4198.3	4672.1
Ejec5	7519.4	6736	5999.4	7882.7	48518.8	3936.5	3507.5	11168.9	3505	3478.8
Media	10616.8	7719.52	9653.24	8058.7	48911.58	6193.68	5160.3	14295.38	4489.5	4478.48
Desv.	10388.813	1994.041	2276.106	612.637	423.840	3280.905	2914.694	3975.213	1554.716	1126.350



Si comparamos este algoritmo con los anteriores, vemos que tiene unos tiempos de ejecución mayores pero aún así el mayor tarda alrededor de 50 ms, que sigue siendo un tiempo razonable. Esto se debe a que tiene que hacer muchas más operaciones de gestión de las memorias a corto y largo plazo.

Por tanto, podemos decir que la mejor ejecución ha sido la segunda ejecución que utiliza la semilla: 8876012

Por el contrario, si comparamos este algoritmo con los anteriores, podemos observar que las soluciones obtenidas han sido más cercanas a los óptimos globales, excepto en la función 4 (Schewefel) que ha empeorado.

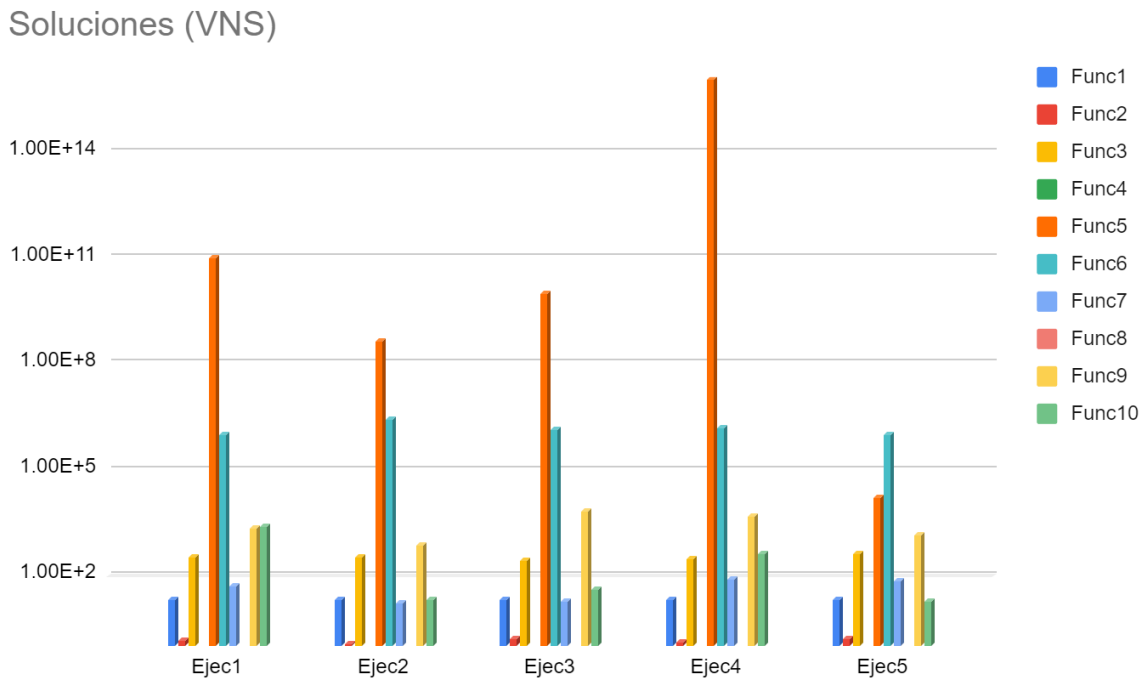
5.3 Búsqueda VNS

Este algoritmo se basa principalmente en el anterior. A diferencia de los anteriores este algoritmo tiene 3 entornos diferentes.

- **Entorno inicial:** Es el mismo que tenemos en BL3 y BLK
- **Entorno aleatorio:** Partimos del mismo operador de generación de vecinos del entorno inicial, pero en este caso si la probabilidad de cambio se cumple generamos un valor completamente aleatorio para esa variable.
- **Entorno cambio signo:** Partimos del mismo operador de generación de vecinos del entorno inicial, pero en este caso si la probabilidad de cambio se cumple invertimos el signo del valor de la variable, y en el caso de Michalewicz hacemos la inversa.

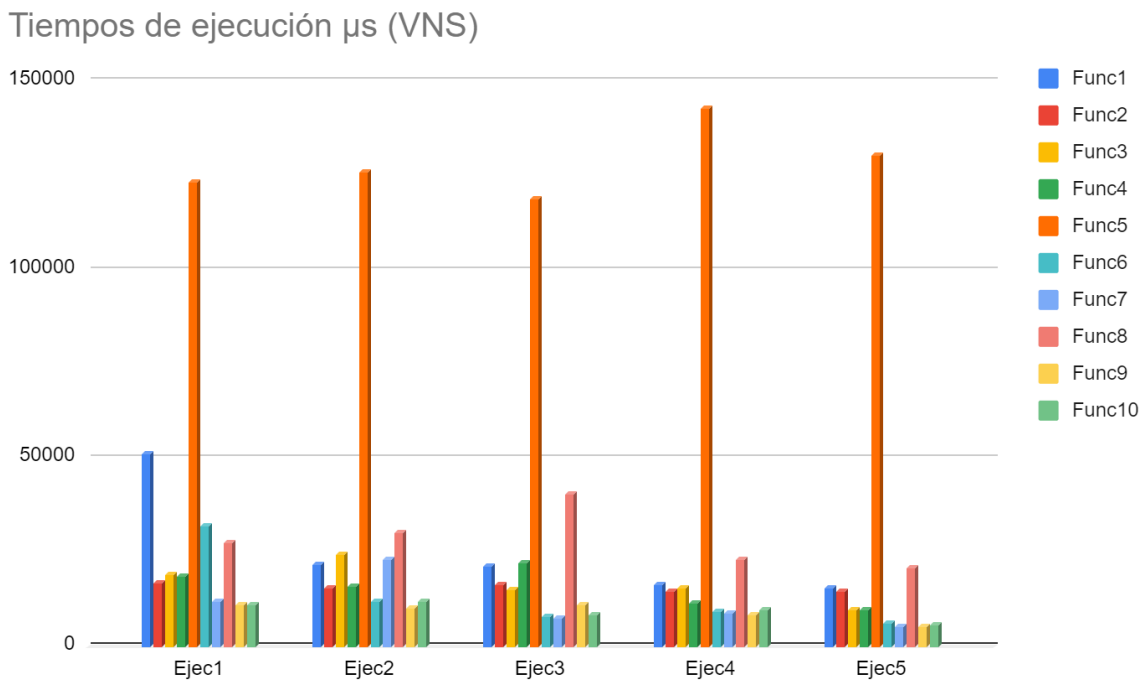
Conforme se ejecuta el algoritmo iremos cambiando de entorno (en order 1, 2, 3 y vuelta a empezar), cuando se produzca un movimiento de empeoramiento.

VNS	**Func1**	**Func2**	**Func3**	**Func4**	**Func5**	**Func6**	**Func7**	**Func8**	**Func9**	**Func10**
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
Ejec1	20.1611	1.3953	315.2203	-29132.4684	9.38E+10	9.05E+05	48.0568	-5.7499	2.06E+03	2328.1469
Ejec2	20.2164	1.1097	330.6341	-28280.3661	4.31E+08	2.57E+06	16.5544	-5.5202	7.00E+02	19.1872
Ejec3	20.1318	1.5813	270.8291	-26557.7398	9.77E+09	1.37E+06	17.4085	-5.9038	6.41E+03	39.4519
Ejec4	20.1688	1.2466	282.8747	-28110.6852	1.05E+16	1.50E+06	80.4297	-5.4599	4.80E+03	414.0559
Ejec5	20.1873	1.6127	389.1105	-31646.7691	16342.0755	9.23E+05	66.0148	-5.3054	1.33E+03	17.8425
Media	20.1731	1.3891	317.7337	-28745.6057	2.10E+15	1.45E+06	45.6928	4.0723	3.27E+03	563.7369
Desv.	0.0314	0.2153	46.5902	1869.0065	4.70E+15	6.80E+05	28.6106	0.2381	2.44E+03	1000.6194



Ahora vamos a ver todos los tiempos de ejecución medidos en microsegundos, para posteriormente hacer un estudio de cuál ha sido la mejor ejecución.

VNS	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
Ejec1	51283.11	16888.86	19127.907	18596.33	123469.18	32125.91	11931.46	27603.42	10931.331	11187.724
Ejec2	21951.77	15692.63	24428.527	16005.46	125844.06	11925.98	22987.30	30081.95	10384.248	11962.723
Ejec3	21345.73	16514.09	15030.503	22310.57	118662.46	8161.70	7341.74	40339.95	11245.45	8430.192
Ejec4	16604.64	14495.90	15592.053	11700.55	142758.87	9234.61	8975.23	23250.01	8457.179	9642.556
Ejec5	15485.33	14690.77	9707.472	9716.23	130342.12	6068.58	5187.29	20684.98	5282.278	5535.744
Media	25334.12	15656.45	16777.2924	15665.83	128215.34	13503.36	11284.60	28392.06	9260.0972	9351.7878
Desv.	14780.90	1064.79	5443.0048	5098.46	9157.57	10622.13	6989.78	7616.92	2473.1602	2533.9554



En este algoritmo también se puede considerar que los tiempos de ejecución son despreciables, aunque sean mayores que los algoritmos anteriores, por tanto, la mejor ejecución es aquella que se acerca más a los óptimos globales de las distintas funciones matemáticas. En este caso ha sido la segunda ejecución con la semilla: 8876012

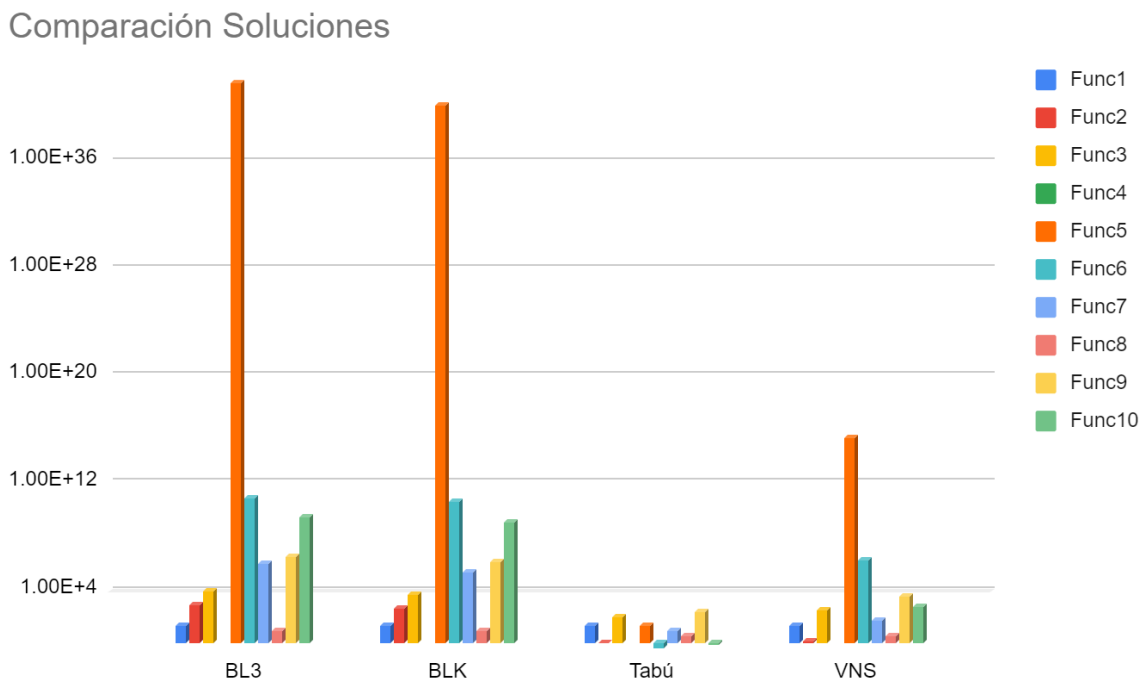
Si comparamos este algoritmo con los anteriores vemos que obtiene mejores resultados que el BL3 y BLK, pero peores que la búsqueda tabú esto seguramente se deba a la definición de los propios entornos.

5.4 Comparación entre algoritmos

Una vez hemos visto el análisis de cada uno de los algoritmos implementados, vamos a comparar los resultados de forma exhaustiva para poder tomar conclusiones y decidir finalmente cual es el algoritmo que obtiene mejores resultados.

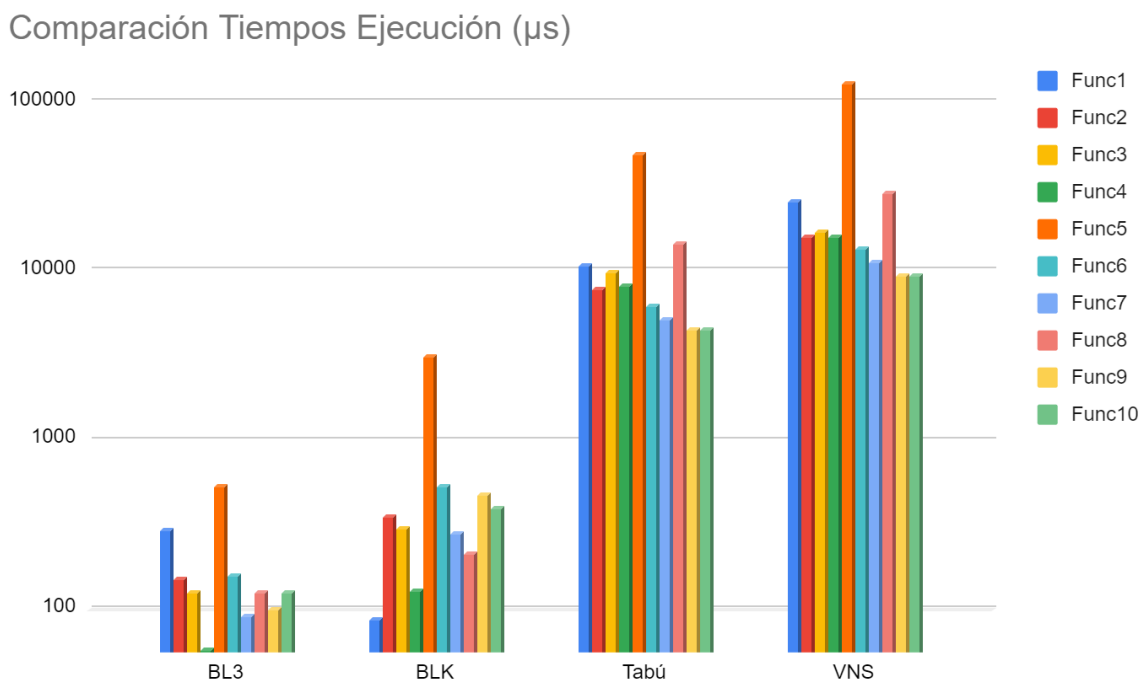
Para ello tal y como hemos hecho anteriormente, vamos a realizar dos tablas. Una con todos los datos de las medias de las desviaciones de error y otra con todas las medias de los tiempos de ejecución por cada uno de los algoritmos, semillas y funciones.

Alg.	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.	Sol.
BL3	21.13184	704.87708	7742.49496	-1301.0843	6.14E+41	6.20E+10	8.70E+05	9.20303	3.08E+06	2.59E+09
BLK	20.71266	434.97228	4721.6582	-11701.915	1.34E+40	3.27E+10	2.00E+05	8.56823	1.29E+06	9.42E+08
Tabú	20.20454	1.0924	101.75906	-20040.8786	2.11E+01	4.05E-01	8.23E+00	3.30673	2.07E+02	7.33E-01
VNS	20.17308	1.38912	317.73374	-28745.60572	2.10E+15	1.45E+06	4.57E+01	4.07231	3.27E+03	5.64E+02



Ahora vamos a hacer la comparación de los tiempos de ejecución para cada uno de los algoritmos, para finalmente obtener conclusiones acerca de cuál es el mejor de ellos.

Alg.	Func1	Func2	Func3	Func4	Func5	Func6	Func7	Func8	Func9	Func10
	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo	Tiempo
BL3	288.92	149.34	123.64	56.8	527.54	156.92	90.74	123.16	98.82	123.16
BLK	86.2	349.86	297.78	127.8	3055.28	520.46	279.14	209.64	468.64	392.26
Tabú	10616.8	7719.52	9653.24	8058.7	48911.58	6193.68	5160.3	14295.38	4489.5	4478.48
VNS	25334.12	15656.45	16777.29	15665.83	128215.34	13503.36	11284.60	28392.06	9260.10	9351.79



Finalmente si comparamos todos los algoritmos con respecto a los tiempos vemos que como máximo el mayor tiempo es alrededor de 100 ms, por tanto, se puede considerar despreciable ya que es una décima parte de un segundo.

Si observamos las soluciones obtenidas (diferencia entre solución obtenida y el óptimo global), el algoritmo que ha conseguido mejores resultados es la búsqueda tabú.

Por tanto, podemos concluir que el mejor algoritmo es la búsqueda tabú, a pesar de ser más simple que el VNS que obtiene peores resultados y en un tiempo de ejecución ligeramente mayor.

6. Enlaces relacionados

- **Archivos de logs:** https://drive.google.com/drive/folders/1Edg78unjYZcZShTxSs97vNl6FPwbeLtS?usp=sharing

- **Optimization Test Functions:** https://www.sfu.ca/~ssurjano/optimization.html